Spack: A Package Manager for Supercomputers, Linux and MacOS

Massimiliano Culpo

SCITAS - EPFL
The complexity of scientific software threatens productivity

- Developers, users and facilities spend a lot of time building and porting scientific software
- Not much standardization in HPC: we are prone to trade portability for performance!
- Target machines may be very different from one another
- Applications need to be tested against different dependencies / compilers to ensure portability

Scientific software must be easy to deploy and use.
What can we do at HPC sites when software is easy to deploy?

Provide to developers:
- Multiple options for the same service (OpenMPI, MVAPICH2, IntelMPI, ...)
- A more extensive choice of compilers
- Easy ways to set-up development environments

Provide to end-users:
- The best tuning of scientific applications
- Convenient ways to use them

And as cluster maintainers:
- We struggle less with installations
- We can explore a broader range of possibilities before deciding what to install for users
- We can easily reproduce a software stack

To reach these goals, we need more automation.
How do we automate things then?
Spack: a package manager for Supercomputers

1. Get Spack:
   
   $ git clone https://github.com/LLNL/spack

2. Set it up:
   
   $ . spack/share/spack/setup-env.sh

3. Install a package:
   
   $ spack install hdf5

Hdf5 and all of its dependencies are downloaded, built and installed within spack directory
What is a package manager?

Spack is a “package manager”:
- Drives build systems, but does not replace them
- Manages dependency DAGs
- Ensures consistent builds

Packages within Spack:
- Can have any build system they want
- Don’t need sources to be modified in any way (pristine sources)

Figuring out configure options takes time:
- Spack is a cache of recipes
- The effort is shared across teams
What makes Spack different from other similar tools?
Spack handles combinatorial software complexity

**Dependency DAG**

- mpileaks
  - callpath
  - dyninst
  - mpi
  - libdwarf
  - libelf

Each graph of dependencies:
- Represents a unique configuration
- Can be mapped to a **unique hash**

**Installation Layout**

- spack/opt/
  - linux-x86_64/
    - gcc-4.7.2/
      - mpileaks-1.1-0f54bf34cadk/
    - intel-14.1/
      - hdf5-1.8.15-lkf14aq3nqiz/
    - bgq/
      - xl-12.1/
        - hdf5-1-8.16-fqb3a15abrwx/
    - ...

Each configuration:
- Is installed in a **unique directory**
- The hash of the graph is appended to the each prefix

Installed packages:
- Have RPATHs embedded in binaries
- No need to set LD_LIBRARY_PATH or similar for Spack built packages
- Things work **the way they were built**
Spack can retrieve the full configuration of installed packages

$ spack find callpath

===> 2 installed packages.
-- linux-x86_64 / clang@3.4 ---- -- linux-x86_64 / gcc@4.9.2 ---------
callpath@1.0.2       callpath@1.0.2

$ spack find -dl callpath

===> 2 installed packages.
-- linux-x86_64 / clang@3.4 ------------------
xv2clz2   callpath@1.0.2
ckjazss ^adep-t-utils@1.0.1
3ws43m4 ^boost@1.59.0
ft7znm6   ^mpich@3.1.4
qqnuet3   ^dyninst@8.2.1
3ws43m4 ^boost@1.59.0
3ws43m4 ^boost@1.59.0
65rdud ^libdwarf@20130729
cj5p5fk   ^libelf@0.8.13
ct5p5fk   ^libelf@0.8.13
65rdud ^libdwarf@20130729
cj5p5fk   ^libelf@0.8.13
ct5p5fk   ^libelf@0.8.13
ft7znm6   ^mpich@3.1.4

Spack stores the information on installed packages in a DB, and can query it at any time
Spack provides a *spec* syntax to describe DAG configurations

```
$ spack install mpileaks
$ spack install mpileaks@3.3
$ spack install mpileaks@3.3 %gcc@5.3.0
$ spack install mpileaks@3.3 %gcc@5.3.0 +threads
$ spack install mpileaks@3.3 cppflags=’-O3’
$ spack install mpileaks@3.3 target=haswell
$ spack install mpileaks@3.3 ^mpich@3.2 %gcc@5.3.0
```

Each expression is a *spec* for a particular configuration:

- Clauses represent constraints to the spec
- Constraints are optional, and you can specify only what you really need
- Installations can be customized on the command line

**Spec syntax is recursive and gives full control over the combinatorial build space**
Spack can constrain versions and options of dependencies

\$ spack install mpileaks %intel@12.1 ^libelf@0.8.12

Spack ensures one configuration of each library per DAG

- Ensures ABI consistency
- Users do not need to know the DAG structure, only the dependency names
- Builds use the same compiler by default, but you can mix them

Spack handles ABI-incompatible, versioned interfaces like MPI

\$ spack install mpileaks ^mvapich@1.9
\$ spack install mpileaks ^mpi@2

Spack permits you to specify an implementation of a virtual dependency, or just the version of its interface
Spack fills in the details when the user is not explicit

The user inputs an abstract spec with some constraints, Spack fully constrains it and passes the concrete configuration to install
Spack builds each package in its own compilation environment.

**Spack process**

- do\_install()
  - Install dep 1
  - Install dep 2
  - ...

**Build process**

- Set up environment
  - CC = spack/env/gcc
  - SPACK\_CC = /usr/bin/gcc
  - CXX = spack/env/g++
  - ...
  - PKG\_CONFIG\_PATH = ...
  - CMAKE\_PREFIX\_PATH = ...
  - PATH = ...

- install()

- Forked build process isolates the environment for each build.
- Compiler wrappers hide part of the complexity.

**Compiler wrappers**

- Add include and libs search paths
- Add RPATH flags
- Ensure dependencies are found
- Load Cray modules
- Injects compiler flags on demand

**Build steps**

- configure
- make
- make install
Spack packages are simple Python scripts

class Dyninst(Package):
    """API for dynamic binary instrumentation."""

    homepage = "https://paradyn.org"

    version('9.3.0', 'edde7847dc673ca69bd59412af572450')
    version('9.2.0', 'ad023f85e8e57837ed9de073b59d6bab')

    depends_on("libdwarf")
    depends_on("boost@1.42:")
    depends_on('cmake', type='build')

    def install(self, spec, prefix):
        with working_dir('spack-build', create=True):
            # Set-up arguments for CMake
            args = ['..',]
            cmake(*args)
            make()
            make("install")
Spack integrates well into different workflows

```
$ pack module refresh -m tcl
```

```
$ module av
--- share/spack/modules/linux-Ubuntu14-x86_64 ---
autoconf-2.69-gcc-4.8-eud4m3w
ghostscript-9.18-gcc-4.8-5n627qp
lua-luaposix-33.4.0-gcc-4.8-s71rtog
autoconf-2.69-gcc-6.1.0-wo3aeho
ghostscript-9.18-gcc-6.1.0-fsencrw
libpciaccess-0.13.4-clang-3.8.0-oqbf3i7
m4-1.4.17-clang-3.8.0-gstiykt
pkg-config-0.29.1-gcc-4.8-6zubacp
automake-1.15-gcc-4.8-z42qh3r
gmp-6.1.1-gcc-4.8-xsezhyp
libpciaccess-0.13.4-gcc-4.8-62psbr2
m4-1.4.17-gcc-4.8-drmfctm
...
```

```
#%Module1.0
## Module file created by spack ...
##
## <full-spec>
##
module-whatis "autoconf @2.69"
proc ModulesHelp { } {
  puts stderr " Autoconf -- system configuration part of autotools"
}
prepend-path PATH
"<full-prefix>/autoconf-2.69-eud4m3weu6hfeu4o22xiwavnrarxj1uo/bin"
prepend-path MANPATH ...
prepend-path CMAKE_PREFIX_PATH ...
```

Spack generates module files or local views to ease the use of installed software.
Spack integrates well into different workflows

\$ spack view symlink <path> <spec>

${HOME}/local_view/
  └── bin
      ├── x86_64-pc-linux-gnu-gfortran -> .../gcc-4.8/gcc-6.3.0-mkau7f7p/bin/x86_64-pc-linux-gnu-gfortran
      └── doc
          └── ansidecl.h -> spack/opt/spack/linux-ubuntu14-x86_64/gcc-4.8/binutils-2.27-l7va7nysnq/include/ansidecl.h

...  

boost
  └── zlib.h -> spack/opt/spack/linux-ubuntu14-x86_64/gcc-6.3.0/zlib-1.2.10-riqxnlm4s/include/zlib.h

lib
  └── engines

...  

libz.so.1.2.10 -> spack/opt/spack/linux-ubuntu14-x86_64/gcc-6.3.0/zlib-1.2.10-riqxnlm4s/lib/libz.so.1.2.10
  └── pkgconfig
      └── libexec
      └── man
      └── share
          └── x86_64-pc-linux-gnu

Spack generates module files or local views to ease the use of installed software.
How can people contribute?
Spack is sustained by its community

Spack is hosted on Github:
- LLNL people are *extremely open* to new contributions
- More than 30 organizations, more than 100 contributors

Spack is already used in production:
- LLNL, ANL, EPFL
- ...

Github permits to manage contributions:
- Gitflow-like development
- Continuous integration, code coverage
- Fork and pull-requests
- Code-reviews, discussions on features
Contributions to Spack have grown rapidly after SC15.
Spack is well documented

**Readthedocs:**
- Generate docs using Sphinx
- Keeps docs in sync
- Old versions are kept
Questions?